A Coalgebraic Semantics of Subtyping

Erik Poll

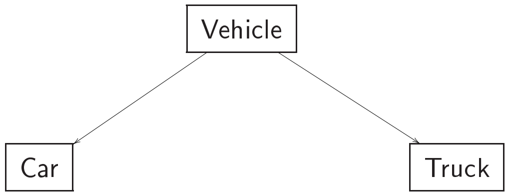
University of Nijmegen

CMCS'2000, 26 March

Coalgebraic Semantics of Objects

Coalgebras can explain some aspects of OO:
hidden internal state, accessible via methods, . . .

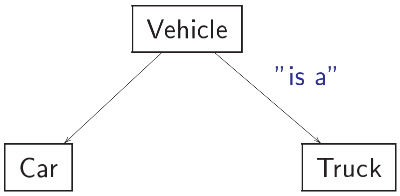
Can coalgebras explain what this means ?



Example class hierarchy

OO background: Meaning of Class Hierarchy

What is this *supposed* to mean ?



A car "is a" vehicle. What does "is a" mean ?

Two **different** interpretations are possible:

- (behavioural) subtyping
- inheritance

OO background: behavioural subtyping = "is a"

Intuition behind "a car is a vehicle":
code written for vehicles also works for cars, or
replacing vehicles with cars does not cause problems.

A necessary condition is **signature subtyping**:
subclass must have all methods of superclass with compatible types.

But for **behavioural subtyping** we want more:
methods must also have the "same" meaning.

Different definitions exist in the OO literature:

(i) in terms of **specifications**:
The objects in the subclass meet a *stronger specification* than objects in the superclass.

(ii) using **substitutability**:
For every object a_{sub} in the subclass there is an object a in the superclass such that *substituting* a_{sub} for a does not affect the semantics.

OO background: inheritance $\neq$"is a"

Inheritance: defining a subclass by adding and/or overriding methods and fields to a superclass.

Overriding methods $\Rightarrow$ subclass may behave completely different than superclass.

So: **inheritance $\not\Rightarrow$ behavioural subtyping !**

Also: no inheritance $\not\Rightarrow$ no behavioural subtyping.

Inheritance vs behavioural subtyping:

- *inheritance* is about inheriting *implementation*
- *subtyping* is about inheriting *behaviour*

If objects are black boxes, then inheritance is about what's inside the boxes. Subtyping is about what's observable on the outside.

Class Specifications

A **class specification** lists methods, assertions, and creation conditions (as in CCSL).

For example:

```
CLASS Counter
  METHODS getcount : X -> Int
              count : X -> X

  ASSERTIONS
   ∀x:X.
     getcount(count(x)) = getcount(x)+1

  CREATION CONDITIONS
     getcount(new Counter) = 0
END
```

The **signature** of a class with methods $m_i : X \to M_i(X)$ is the functor $M(X) = M_1(X) \times \ldots M_n(X)$.

The signature of `Counter` *is* $M(X) = \mathtt{Int} \times X$.

Semantics of Class Specifications

A **model** of a class spec is an M**-coalgebra**

$$(S, \langle m_1, \ldots, m_n \rangle) : X \to M(X)$$

such that the assertions hold and there in an initial state $init \in S$ for which the creation conditions hold.

For example, $(Nat, \langle id_{Nat}, Succ \rangle)$ *is a model of* `Counter`.

Assertions and Creation Conditions

Some (natural) **restrictions**:

- assertions universally quantified over state space
  `∀ x:X. ...` ,
  specifying a property of each individual object.

  So if (S_1, m_1) *and* (S_2, m_2) *are models, then* $(S_1 + S_2, [M(inl) \circ m_1, M(inr) \circ m_2])$ *is a model.*

- assertions should use bisimilarity, not equality on the state space

  E.g. not `count(x)` $\neq$ `x`, but `count(x)` $\overset{\leftrightarrow}{\neq}$ `x`.

  So if (S_1, m_1) *is a model and there is a total bisimulation between* (S_1, m_1) *and* (S_2, m_2)*, then* (S_2, m_2) *is a model.*

Signature Subtyping

A' is a **signature subtype** of A if it has **more methods**

For example:

```
CLASS RCounter
  METHODS getcount : X -> Int
             count : X -> X
             reset : X -> X
        ⋮
END
```

is a signature subtype of

```
CLASS Counter
  METHODS getcount : X -> Int
             count : X -> X
        ⋮
END
```


Semantics of Signature Subtyping

Signature subtyping between classes yields a **natural transformation** η between their signatures.

This η induces a functor from M'- to M-coalgebras, namely

$$(S, m') \mapsto (S, \eta \circ m')$$

For example, for RCounter *and* Counter*,*

$$\eta = \langle \pi_1, \pi_2 \rangle : \mathtt{Int} \times X \times X \to \mathtt{Int} \times X$$

and the associated functor is

$$(S, \langle imp_{getcount}, imp_{count}, imp_{reset} \rangle) \\ \mapsto \\ (S, \langle imp_{getcount}, imp_{count} \rangle)$$

Behavioural Subtyping

A' is a **behavioural subtype** of A, A'$\leq_{assert}$ A, iff it has **more methods** and **stronger assertions**.

For example: RCounter *is a behavioural subtype of* Counter*, where*

```
CLASS RCounter
  METHODS getcount : X -> Int
             count : X -> X
             reset : X -> X
  ASSERTIONS
   ∀ x:X.
        getcount(count(x)) = getcount(x)+1
   ∀ x:X.
        getcount(reset(x)) = 0

  CREATION CONDITIONS
     getcount(new Counter) = 0
END
```


Semantics of Behavioural Subtyping

How to define behavioural subtyping between coalgebras ?

Liskov's substitution principle: For every object a_{sub} in the subclass there is an object a in the superclass such that substituting a_{sub} for a in a program does not change its meaning.

Given $\eta : M' \to M$, an M'-coalgebra (S', m'), an M-coalgebra (S, m).

(S', m') is a **behavioural subtype** of (S, m), $(S', m') \leq_\eta (S, m)$, iff

$$\forall s' \in S'.\ \exists s \in S.\ s' \sim s$$

for some M-**bisimulation** $\sim$ between $(S', \eta \circ m')$ and (S, m).

Semantics of Behavioural Subtyping

Let $behaviour_m : (S, m) \to (\nu M, \alpha)$ be the mapping from an M-coalgebra (S, m) to the final M-coalgebra $(\nu M, \alpha)$.

Theorem

$$(S', m') \leq_\eta (S, m)$$
$$\iff$$
$$behaviour_{\eta \circ m'}(S') \subseteq behaviour_m(S)$$

So, for example, $(\mathit{Car}, m') \leq_\eta (\mathit{Vehicle}, m)$ iff the set of all Car-behaviours is a subset of the set of all Vehicle-behaviours.

Soundness & Completeness

Connection between

- $\leq_\eta$ on coalgebras, and
- $\leq_{assert}$ on coalgebraic specifications ?

Theorem

A' is a behavioural subtype of A
$\iff$
$\forall (S', m') \models A'.\ \exists (S, m) \models A.\ (S', m') \leq_\eta (S, m).$

provided A' is consistent.

This theorem relies on the restrictions imposed on assertions earlier.

Coalgebraic Refinement

Refinement is a central notion in specification languages:

A' **refines** A iff any model of A' can be "turned into" a model of A.

In other words, A' refines A if any model of A' can be used to mimic the behaviour specified by A.

For example, `RCounter` is a refinement of `Counter`.

What's the relation between refinement and subtyping?

Subtyping $\neq$ Coalgebraic Refinement

`DCounter` is a refinement, but not a behavioural subtype of `Counter`

```
CLASS DCounter
 METHODS getcount : X -> Int
            count : X -> X
         getdelta : X -> Int
         setdelta : X × Int -> X

 ASSERTIONS
   getcount(count(x)) = getcount(x)+getdelta(x)
   getcount(setdelta(x,n)) = getcount(x)

   getdelta(setdelta(x,n)) = n
   getdelta(count(x)) = getdelta(x)

 CREATION CONDITIONS
   getcount(new DCounter) = 0
   getdelta(new DCounter) = 1
END
```

Subtyping vs Coalgebraic Refinement

Lemma

- If A' is a refinement of A then

$$\forall (S', m') \models A' \; \exists (S, m) \models A \\ behaviour_{\eta \circ m'}(S') \supseteq behaviour_m(S)$$

- If A' is a behavioural subtype of A then

$$\forall (S', m') \models A' \; \exists (S, m) \models A \\ behaviour_{\eta \circ m'}(S') \subseteq behaviour_m(S)$$

Refinements have more behaviour, subtypes have less.

CONCLUSIONS

Coalgebras provide a nice interpretation for (behavioural) subtyping, both

- between coalgebras (classes)
- between coalgebraic specifications (class specifications)

Behavioural subtyping $\begin{array}{c}\not\Rightarrow \\ \not\Leftarrow\end{array}$ coalgebraic refinement

Coalgebras do *not* provide a nice interpretation for inheritance with late binding (because coalgebras do not explain self-references).